Table des matières

1. Exemple d'utilisation de malloc, free, et de passage par adresse 1
2. Les signaux.....2
Comment et quand sont envoyés les signaux 3
L'appel système kill 3
Paramètres 3
Résultat 3
Variante : la fonction raise 3
Note 3
La commande UNIX kill 4
Certains caractères au terminal 4
Certaines conditions hardware 4
Certaines conditions software 4
Que fait le processus avec un signal reçu : l'appel système signal 4
Fonction 4
Paramètres 5
Résultat 5
Exemple 5
Autres fonctions relatives aux signaux 5
Le PIC.....6
L'environnement hardware.....6
Le microcontrôleur PIC18F452 6
La mémoire de données 6
La mémoire de programme 6
Le temporisateur 6
Le LCD 7
Le RPG 7
Le port série 7
L'environnement software 7
Environnement software du microcontrôleur 7
exercice de sensibilisation à l'impact de la hiérarchie des mémoires.....7

1. Exemple d'utilisation de malloc, free, et de passage par adresse

```
 * à une fonction d'un argument de type 'pointeur'.
 ************************************************************************/

#include <stdio.h>
#include <stdlib.h>

/* Protoype de la fonction */
void alloue (int nb, int **memoire);

int main ()
{
  int i, *valeurs, nb;
  int puissance2;

  printf ("Entrez le nombre d'éléments : ");
  scanf ("%d", &nb) ;

  /* Alloue une zone mémoire pouvant contenir 'nb' éléments de
   * type entier. Comme 'valeurs' est de type 'pointeur vers un
```

```
   * entier', donc '&valeurs' est l'adresse d'un pointeur vers
   * un entier, c'est-à-dire un 'pointeur vers un pointeur vers
   * un entier'. De la sorte, 'valeurs' pourra être modifié par
   * la fonction 'alloue'.
   */
  alloue (nb, &valeurs);

  /* On peut accéder au éléments de la mémoire comme pour un
   * tableau. On calcule les puissances de 2.
   */
  puissance2 = 1;
  for (i = 0; i < nb; i++) {
    valeurs[i] = puissance2;
    puissance2 *= 2;
  }

  /* Affichage du tableau */
  for (i = 0; i < nb; i++) {
    printf("2^%d = %d\n", i, valeurs[i]);
  }

  /* Libère la mémoire */
  free(valeurs);

  return 0;
}
/*****************************************************************
 * Fonction : alloue
 * PRE : 'nb' est le nombre d'entiers pour lesquels il faut
 *       allouera la zone mémoire. 'nb' >= 0.
 * POST: - si l'allocation réussit : 'memoire' pointe vers le début
 *         d'une zone mémoire pouvant contenir 'nb' entiers.
 *       - sinon le programme se termine.
 *****************************************************************/
void alloue (int nb, int **memoire)
{
  /* Allocation de la mémoire */
  *memoire = (int *) malloc( nb * sizeof(int) );

  if (*memoire == NULL) {
    printf("Pas assez de mémoire !\n");
    exit(-1);
  }
}
```

2. Les signaux

Un signal est une notification à un processus qu'un événement s'est produit. Les signaux sont parfois appelés ``interruptions software''. La plupart des signaux sont asynchrones, c'est à dire que le processus ne sait pas exactement quand un signal sera généré. Mais les signaux peuvent être générés de façon synchrone lorsqu'une erreur se produit dans une application (par exemple le signal `SIGSEGV` ``Segmentation Fault'').

Les signaux peuvent être envoyés à un processus lorsque le système détecte un événement software, par exemple un utilisateur entrant une requête d'interruption ou d'arrêt depuis un autre processus. Les signaux peuvent aussi venir directement du système d'exploitation. De même, lorsqu'un processus se termine anormalement, il essaie généralement d'envoyer un signal indiquant ce qui s'est passé. Les programmes C peuvent les attraper dans un but de diagnostique par exemple.

En résumé, les signaux peuvent venir :

- d'un autre processus,

- du terminal,
- du processus lui-même,
- du système.

Le système définit un ensemble de signaux qui peuvent être envoyés à un processus. Pour utiliser les signaux, il faut inclure la librairie `<signal.h>`. Les signaux peuvent être numérotés de 0 à 31. Quelques signaux (voir `man -s3HEAD signal` ou `man 7 signal`):

```
SIGHUP   1  /* hangup */
SIGINT   2  /* interrupt */
SIGQUIT  3  /* quit */
SIGILL   4  /* illegal instruction */
SIGABRT  6  /* used by abort */
SIGKILL  9  /* hard kill */
SIGALRM 14  /* alarm clock */
SIGTERM 15  /* terminated */
SIGCONT 19  /* continue a stopped process */
SIGCHLD 20  /* to parent on child stop or exit */
```

Remarque : Un seul signal de chaque type peut être enregistré pour chaque processus. Donc, si un processus reçoit plusieurs signaux du même type avant de traiter le premier, seulement un seul signal sera effectivement traité ! Tous les autres signaux sont perdus. De plus, un processus ne peut recevoir de signal que s'il est en cours d'exécution ou en attente d'entrées-sorties.

Comment et quand sont envoyés les signaux

L'appel système kill

L'appel système kill permet à un processus d'envoyer un signal à un autre processus ou à lui-même. Le mot "kill" est mal choisi, puisque le signal ne termine pas forcément le processus : certains signaux ``tuent'' le processus, tandis que d'autres signaux informent seulement le processus d'un événement que ce processus doit gérer.

Pour envoyer un signal, le processus qui envoie et le processus qui reçoit doivent avoir le même propriétaire effectif, ou alors, le propriétaire du processus qui envoie doit être le super user.

Paramètres

`pid`	ID du processus ou du groupe de processus de destination auquel le signal doit être envoyé.
`sig`	numéro de signal, défini dans `<signal.h>`.

Résultat

Renvoie la valeur 0 en cas de succès. -1 en cas d'erreur et la variable `errno` est modifiée en conséquence.

Variante : la fonction raise

```
#include <signal.h>

int raise(int sig);
```

Cette fonction envoie le signal `sig` au programme en cours d'exécution. Elle est l'équivalent de : `kill(getpid(), sig);`

Note

- Si `pid` vaut zéro, le signal est envoyé à tous les autres processus dont l'ID de groupe est égal à l'ID de groupe du processus qui envoie le signal.

- Si `pid` vaut `-1`, le signal est envoyé à tous les autres processus appartenant au même utilisateur.
- Si `sig` vaut zéro, aucun signal n'est réellement envoyé, mais une vérification d'erreur est réalisée (validité du `pid`).

Un processus peut connaître son propre `pid` grâce à la fonction `getpid()`.

La commande UNIX kill

Cette commande envoie le signal indiqué en argument aux processus mentionnés. Si on ne précise pas de signal, le signal `SIGTERM` est envoyé. Ce dernier tuera les processus qui ne l'interceptent pas.

Certains caractères au terminal

Le caractère d'interruption `CTRL-C` termine un processus qui tourne, en générant un signal `SIGINT`. Le caractère de terminaison (souvent `CTRL-\`) termine un processus qui tourne, et provoque un ``core'' (image mémoire) du processus par la génération du signal `SIGQUIT`. Le core peut alors être utilisé pour débugger le programme. En plus de ces deux caractères, certains systèmes en fournissent d'autres. Par exemple, `CTRL-Z` génère un signal `SIGTSTP` (stop).

Certaines conditions hardware

Par exemple :

- Les erreurs d'arithmétique flottante génèrent une erreur `SIGFPE`.
- Référencer une adresse en dehors de l'espace d'adressage d'un processus génère un signal `SIGSEGV` (segmentation fault).

Certaines conditions software

Exemple : le signal `SIGURG` (urgent condition on socket) est généré quand des données ``out of band'' arrivent sur un socket.

Que fait le processus avec un signal reçu : l'appel système signal

Un processus peut ignorer un signal. Tous les signaux, excepté `SIGKILL`, peuvent être ignorés. Un processus peut aussi choisir l'action par défaut pour le signal. La plupart des signaux provoquent l'arrêt du processus si aucune action n'est prise par le processus en réponse à ce signal. Chaque signal possède une action par défaut qui est l'une des suivantes :

- Le signal est ignoré après sa réception
- Le processus est arrêté après la réception du signal
- Un fichier ``core'' est écrit, puis le processus est terminé
- Le processus est stoppé après la réception du signal

Un processus peut aussi définir lui-même le comportement à adopter lors de la réception d'un signal. Il suffit de spécifier avant que l'événement se produise la fonction à appeler lors de la réception du signal. Cette fonction est nommée ``signal handler''.

Un processus spécifie comment il désire que le signal soit géré en appelant la fonction `signal()` :

Fonction

L'appel-système `signal()` installe un nouveau gestionnaire pour le signal numéro `sig`. Le gestionnaire de signal est `func` qui peut être soit une fonction spécifique de l'utilisateur, soit l'une des constantes `SIG_IGN` ou `SIG_DFL`.

Lors de l'arrivée d'un signal correspondant au numéro `sig`, les évènements suivants se produisent : Si le gestionnaire correspondant est configuré avec `SIG_IGN`, le signal est ignoré. Si le gestionnaire vaut `SIG_DFL`, l'action par défaut pour le signal est entreprise. Finalement, si le gestionnaire est dirigé vers une fonction `func`,

alors tout d'abord le gestionnaire est re-configuré à `SIG_DFL`, ou le signal est bloqué, puis func() est appelé avec l'argument `sig`.

Paramètres

`sig`	spécifie the numéro du signal (défini dans `<signal.h>`)
`func`	l'action à prendre : `SIG_DFL` : action par défaut (dépend du signal) `SIG_IGN` : signal ignoré (inapplicable pour `SIGKILL`) une fonction : capture le signal et exécute cette fonction avec le numéro du signal comme argument (inapplicable pour `SIGKILL`).

Résultat

Retourne la valeur précédente de `func` pour le signal spécifique `sig`. En cas d'erreur, la valeur `-1` est retournée et la valeur `errno` est modifiée en conséquence.

Exemple

On voudrait être sûr que les fichiers temporaires soient effectivement effacés, même lorsqu'un processus se termine prématurément à cause d'un signal `SIGINT`.

```
#include <signal.h>

void introut(int);

int main()
{
  /* Appelle la fonction introut lorsque le signal SIGINT sera
   * généré, seulement si ce signal SIGINT n'est pas actuellement ignoré
   */
  if (signal(SIGINT, SIG_IGN)!=SIG_IGN)
     signal(SIGINT, introut);
     ...
  creat(tempfile, mode);
     ...
  unlink(tempfile);  /* Fin normale */
  exit(0);
}

void introut(int sig) {
  unlink(tempfile);  /* Nettoyage */
  exit(1);
}
```

Autres fonctions relatives aux signaux

alarm

La fonction `alarm(unsigned int sec)` envoie le signal `SIGALRM` au processus lui-même après `sec` secondes. Voir `man alarm`.

pause

Met le processus en attente jusqu'à ce qu'il reçoive un signal. Voir `man pause`.

sleep

Équivalent à :

```
  signal(SIGALRM, nothing);
  alarm(sec);
  pause();
    ...
```

avec :

```
void nothing(int sig) {
  signal(SIGALRM, SIG_DFL);
}
```

Le PIC

L'environnement hardware

Le système sur lequel vous allez travailler est constitué principalement d'un microcontrôleur (la grande puce rectangulaire dans la figure 1), d'un écran LCD, d'un RPG ("rotary pulse generator", en bas, près de l'écran),d'un port série (à gauche dans l'image) et d'un software button. (tout en bas).Le système fonctionne avec une alimentation de 9V (le connecteur est tout à gauche).

Le microcontrôleur PIC18F452

Un microcontrôleur concentre en une seule puce toutes les fonctionnalités d'un petit ordinateur. Il comporte une unité d'exécution (le "processeur"), une mémoire RAM, FLASH et EEPROM, des périphériques tels qu'un timer (une horloge) ainsi qu'un ou plusieurs ports pouvant être utilisés, par exemple, comme ports série ou parallèle.

Le microcontrôleur que nous allons utiliser pour ce projet est le 18F452 de la firme Microchip. Un diagramme bloc de ce microcontrôleur est illustré par la figure 2. Ce microcontrôleur présente une architecture Harvard (mémoires distinctes pour programme et données). Les données sont placées dans une mémoire de type RAM, d'une capacité de 1536 octets. La mémoire de programme est constituée de mots de 16 bits, est de type FLASH (non volatile) et a une capacité de 2^{14} = 16 K mots (donc 32 Koctets). Ces ressources sont donc précieuses, en comparaison de celles des ordinateurs classiques. Le 18F452 possède encore 5 ports (A à E) et 3 temporisateurs ("timers"' on y reviendra).

La machine dispose d'une petite pile (stack) interne utilisée pour les appels de procédure, mais elle n'a que 31 niveaux. Il ne faudra donc pas dépasser ce niveau d'imbrication d'appels, sous peine d'obtenir un comportement imprévisible.

La mémoire de données

La mémoire de données apparaît comme une série de registres. Le microcontrôleur dispose de deux types de mémoire de données : une mémoire RAM (volatile) et une mémoire EEPROM (non volatile) de **256 Bytes**. La mémoire RAM est divisée en 15 bancs de 256 registres, mais seuls 6 de ces bancs contiennent vraiment de la RAM (de même un PC a un espace d'addressage de 4 Gbytes, mais a rarement plus d'un Giga de mémoire RAM réelle), les bancs 0 à 5. Certains sont des registres spécialisés (les *Special Function Registers* du banc 15), d'autres sont des registres d'usage général (les *General Purpose Registers*). Les registres spécialisés sont des registres liés aux périphériques ou aux registres d'état du microcontrôleur. La figure 3 détaille l'ensemble des registres et la manière de choisir les bancs au moyen du registre spécialisé BSR et du bit a.

La mémoire de programme

Le PIC 18F452 dispose de 16K fois 16 bits d'espace mémoire de programme. Il s'agit d'une mémoire de type FLASH.

Le temporisateur

Le principe d'un temporisateur est celui d'un registre qui est incrémenté régulièrement et automatiquement (à chaque coup d'horloge par exemple), et qui, lors d'un débordement, génère une interruption. Nous vous fournissons en annexes les feuilles du manuel d'utilisation concernant le premier temporisateur du PIC (le timer0).

Comme le temporisateur produit une interruption, nous vous fournissons également la description d'un registre spécial : le registre *intcon*, décrit dans la figure 5. Ce registre sert à la gestion des interruptions. Il permet d'activer ou non toutes ou seulement certaines interruptions. Il permet également de tester quelle interruption est produite.

Question : quels sont les bits en relation avec l'utilisation du timer0 ?

Le LCD

Le LCD qui sera utilisé pour ce projet comporte 2 lignes de 8 caractères. Le LCD se commande au moyen d'instructions qui lui sont spécifiques. Ces instructions sont par exemple l'effacement de l'écran, le déplacement du curseur ou l'écriture d'un caractère. Chacune de ces opérations prend un certain temps, qui varie selon l'opération effectuée (par exemple : 1,64 millisecondes pour effacer l'écran, 40 microsecondes pour déplacer le curseur d'une position). Toutes ces opérations et leurs caractéristiques sont décrites dans le tableau de la figure 6.

Le RPG

Il est utilisé pour incrémenter ou décrémenter une valeur.

Le port série

Il est utilisé pour communiquer avec l'ordinateur. Ceci permet une programmation du PIC directement depuis un PC. Il permet aussi de monitorer le fonctionnement du PIC (voir environnement software).

L'environnement software

La programmation du PIC se fera dans un langage C Les étapes nécessaires pour la mise en route de votre programme sont les suivantes :

1. écriture du programme en C,
2. compilation,
3. transfert du programme vers le microcontrôleur,
4. lancement du programme sur le PIC,
5. monitoring du fonctionnement, tests et validation du programme.

Environnement software du microcontrôleur

Vous allez programmer sur ``machine nue'', vous aurez accès sans contrôle à tous les registres et toutes les ressources du microcontrôleur. Nous vous fournissons cependant un mini-système d'exploitation (QwikBug) qui a les rôles suivants :

- il se charge de la réception de votre programme depuis le PC et son enregistrement dans la mémoire FLASH du PIC,
- il initialise le microcontrôleur et les périphériques lors de la mise sous tension du système,

Bien que vous ayez théoriquement accès à toutes les ressources du microcontrôleur, en pratique, le compilateur a été configuré afin que votre programme n'interfère pas avec le système d'exploitation, du moins si vous vous limitez à programmer en C. Sinon, aucune garantie n'est donnée !

exercice de sensibilisation à l'impact de la hiérarchie des mémoires

La cache et la mémoire virtuelle sont des facilités très pratiques sur les ordinateurs actuels. Elles permettent de cacher au programmeur la complexité de la gestion hardware de la mémoire. Dans la plupart des cas, le programmeur peut les ignorer. Cependant, en pratique, les performances des programmes qui manipulent de gros volumes de données en mémoire pourront dépendre fortement de l'habileté du programmeur à utiliser efficacemment la hiérarchie des mémoires.

L'objectif de ce problème pratique est de vous sensibiliser à l'impact de la hiérarchie de mémoire sur l'exécution et les performances des programmes applicatifs. Pour ce faire, vous écrirez un programme C qui remplit un tableau dont la taille est un peu inférieure à la cache de votre ordinateur, puis qui fait la même chose pour un tableau 2 fois plus grand, puis 3 fois,...25 fois.

1. Compilez deux variantes de ce programme. La première avec les options standard du compilateur C et la

seconde avec l'option -static. Discutez la différence de taille entre ces deux exécutables et analysez brièvement l'état de la mémoire du programme lorsqu'il s'exécute (utilisez la commande "pmap -x <pid>").

2. Evaluez la performance de votre programme lorsque votre tableau tient juste dans la cache, est 2 fois plus grand, 3 fois, etc.; comparez les résultats avec un programme dont le tableau ne change pas de taille mais est rempli 2 fois, 3 fois, etc.

Remarques:

- Pour effectuer des mesures de performances utilisables, il est important que chaque groupe fasse toutes ses mesures sur la même machine et que cette machine soit la moins chargée (RAM et CPU) possible durant les mesures. N'oubliez pas que vous pouvez vous connecter sur ces machines via ssh plutôt que d'être directement à la console. Il n'est pas inutile de reproduire quelques fois chaque mesure afin de vérifier qu'il n'y a pas eu d'interférences durant la mesure
- A titre d'information, voici les caractéristiques techniques des trois principaux types de machines auquelles vous avez accès :
    - Salle Parnas: Sun's Ultra 5, 64Mo RAM, 16ko L1 d-cache, 16ko L1 i-cache, 256ko L2 cache
    - Salle Siemens: Pentium 4, 248Mo RAM (ou plus: variable d'une machine à l'autre), 12kμops trace cache, 8ko L1 d-cache, 512ko L2 cache
    - Salle Intel Pentium 3, 128Mo RAM, 16ko L1 d-cache, 16ko L1 i-cache, 256ko L2 cache
- La fonction suivante peut vous servir à évaluer les temps d'exécution et l'activité de la mémoire virtuelle.

```
void print_usage()
{
  int errno;
  struct rusage usage; /* see sys/resource.h */

  if ((errno=getrusage(RUSAGE_SELF, &usage))==0)
    {
      /* Only a subset of getrusage is supported by Linux 2.4 */

      printf("User time : %d.
%d\n",usage.ru_utime.tv_sec,usage.ru_utime.tv_usec);
      printf("System time : %d.
%d\n",usage.ru_stime.tv_sec,usage.ru_stime.tv_usec);
      printf("Page reclaims : %ld\n",usage.ru_minflt);
      printf("Page faults : %ld\n",usage.ru_majflt);

      printf("Swaps : %ld\n",usage.ru_nswap);
    }
  else
    {
      printf("Error with getrusage\n");
    }
}
```